

Lecture 6

Register allocation

Motivation

Normal form is convenient for intermediate code.

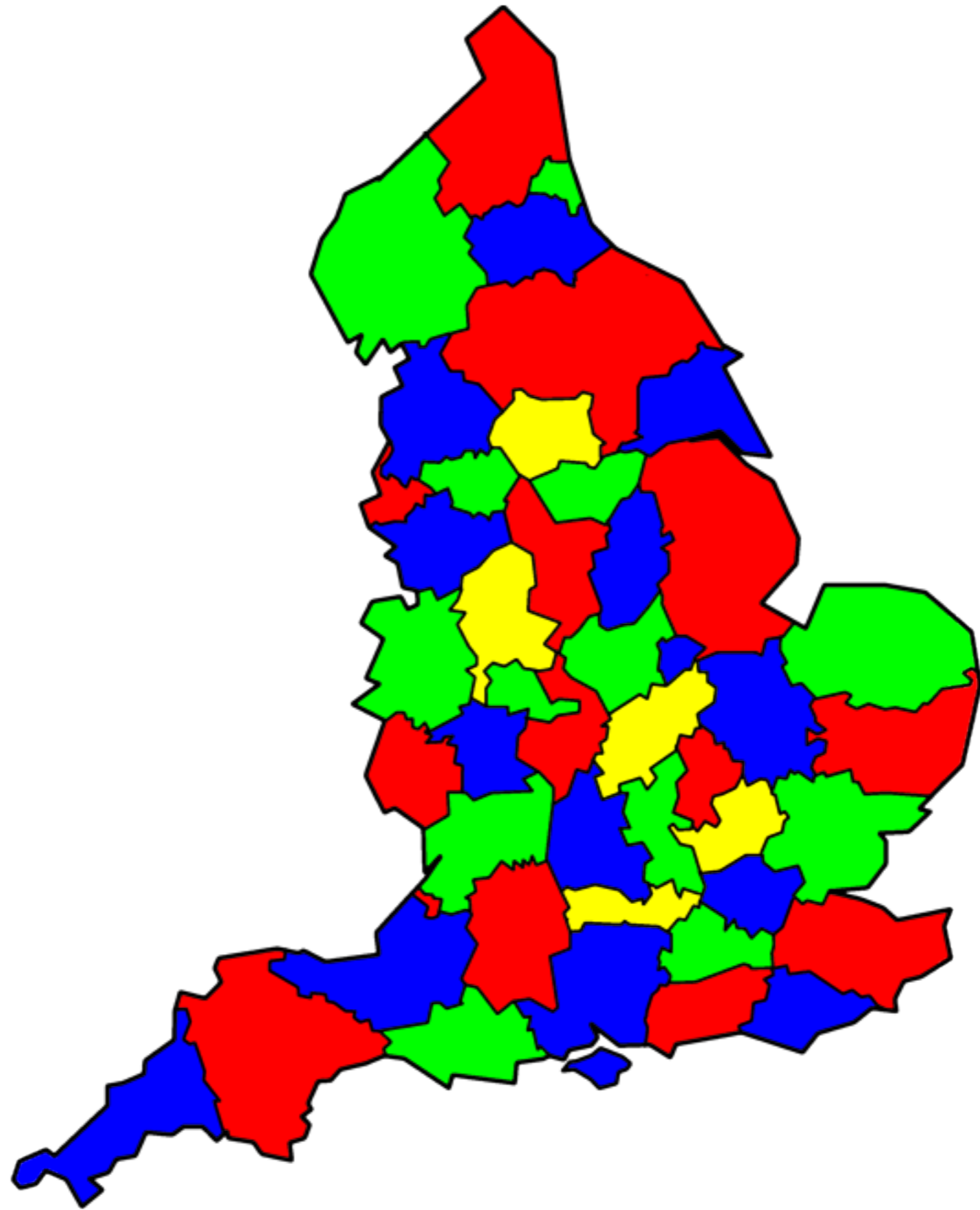
However, it's extremely wasteful.

Real machines only have a small finite number of registers, so at some stage we need to analyse and transform the intermediate representation of a program so that it only requires as many (architectural) registers as are really available.

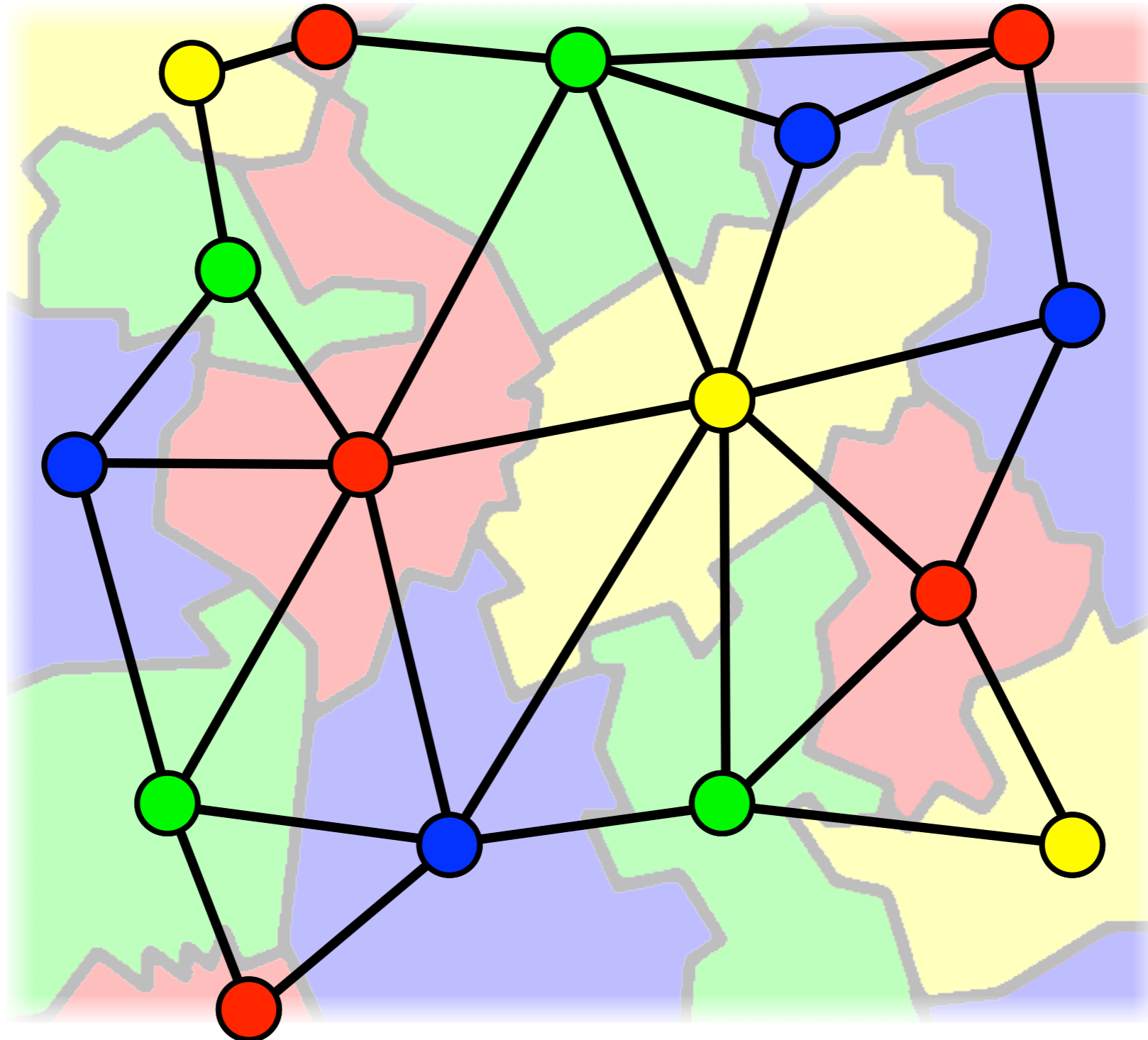
This task is called *register allocation*.

Graph colouring

Register allocation depends upon the solution of a closely related problem known as *graph colouring*.



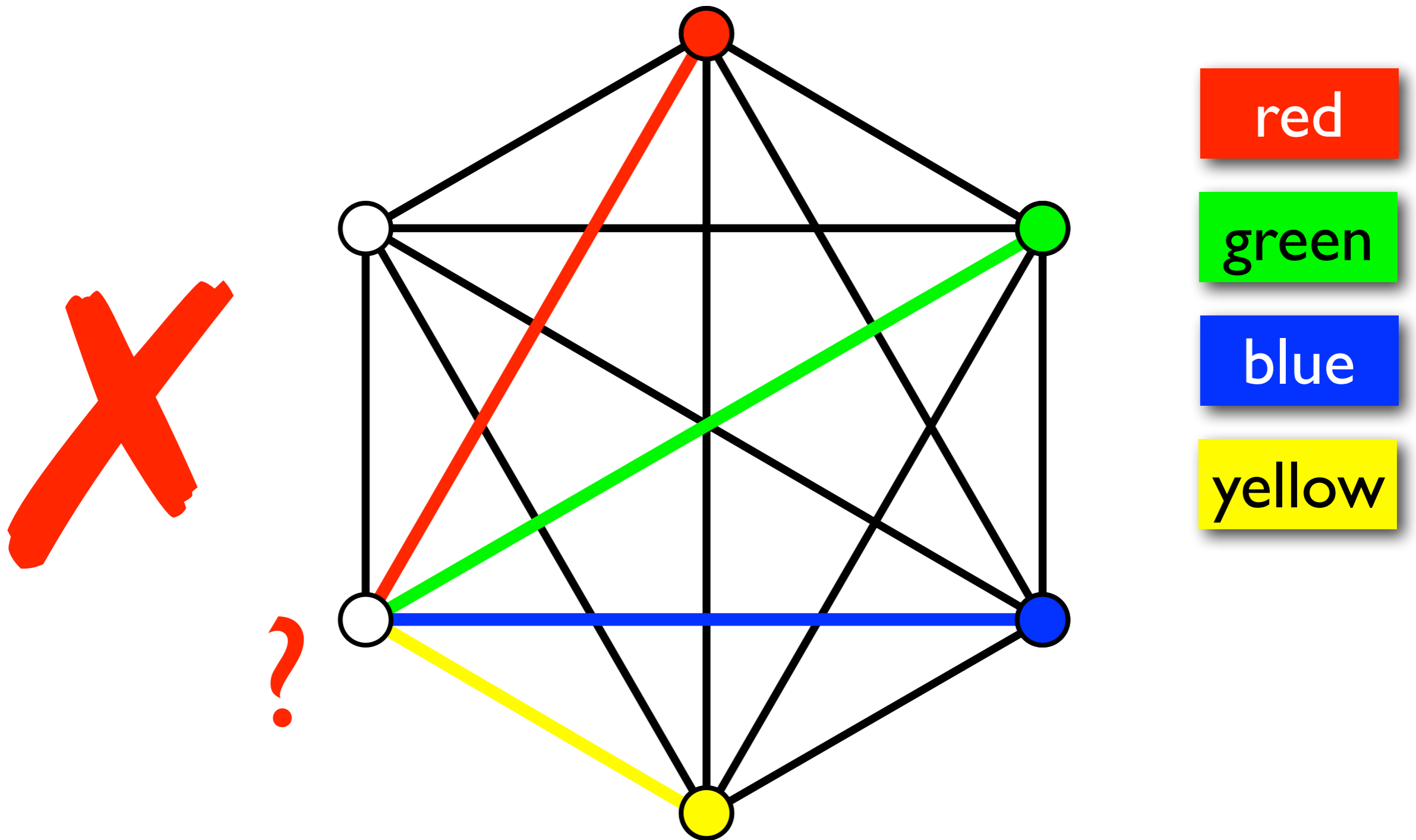
Graph colouring



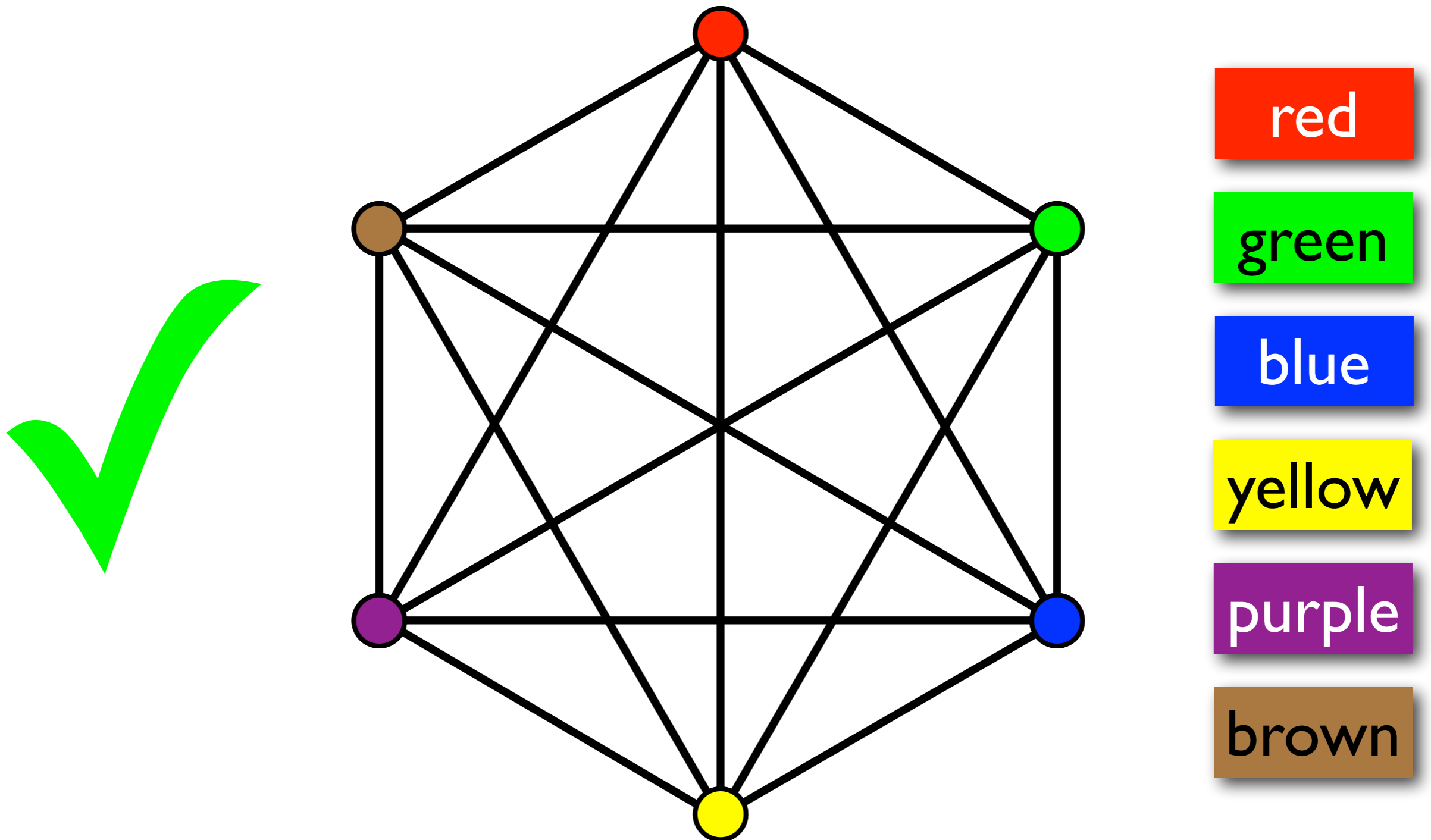
Graph colouring

For general (non-planar) graphs, however, four colours are not sufficient; there is no bound on how many may be required.

Graph colouring



Graph colouring



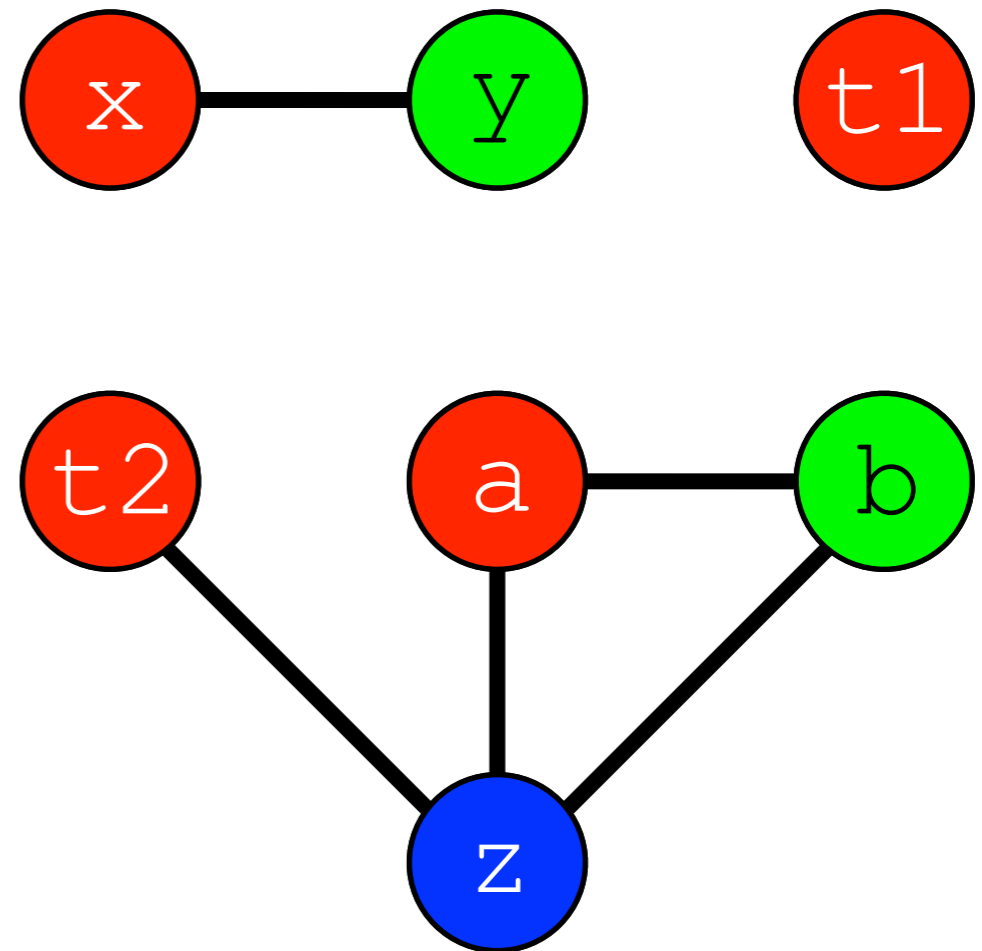
Allocation by colouring

This is essentially the same problem that we wish to solve for clash graphs.

- How many colours (i.e. *architectural* registers) are necessary to colour a clash graph such that no two connected vertices have the same colour (i.e. such that no two simultaneously live virtual registers are stored in the same arch. register)?
- What colour should each vertex be?

Allocation by colouring

```
MOV  r0, #11
MOV  r1, #13
ADD  r0, r0, r1
MUL  r2, r0, #2
MOV  r0, #17
MOV  r1, #19
MUL  r0, r0, r1
ADD  r2, r2, r0
```



Algorithm

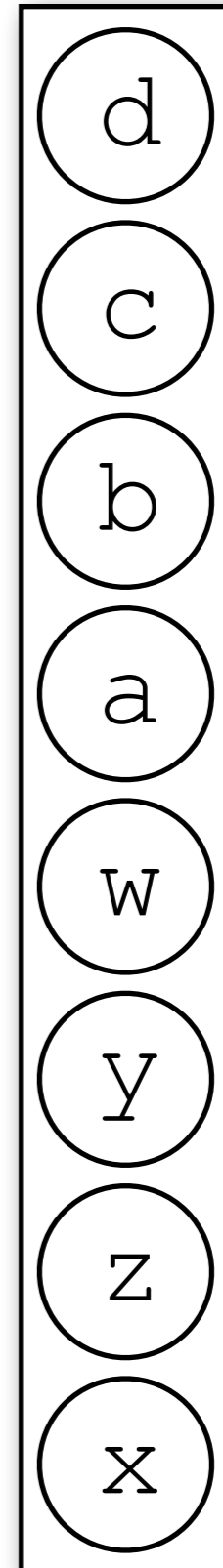
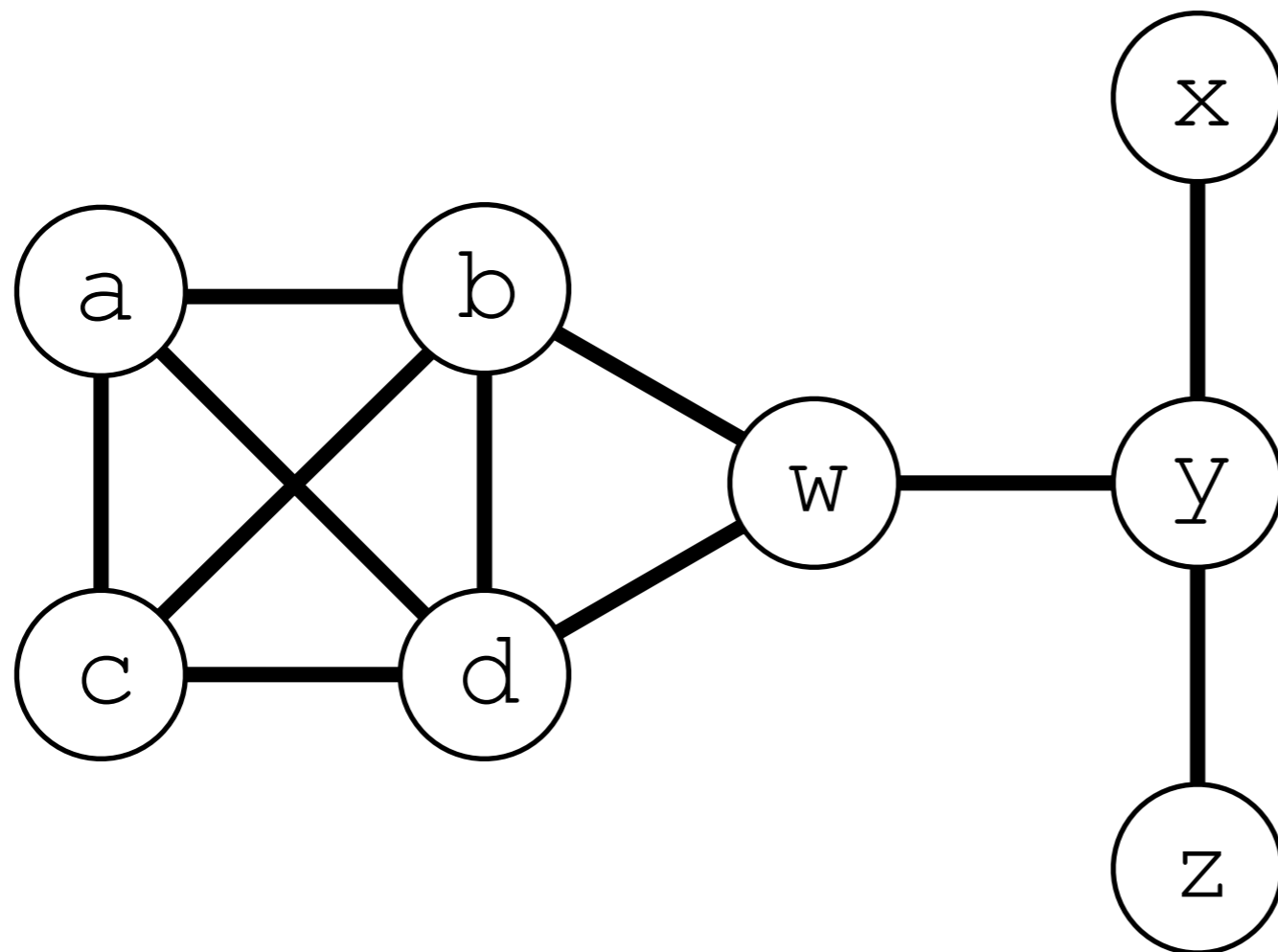
Finding the minimal colouring for a graph is NP-hard, and therefore difficult to do efficiently.

However, we may use a simple heuristic algorithm which chooses a sensible order in which to colour vertices and usually yields satisfactory results on real clash graphs.

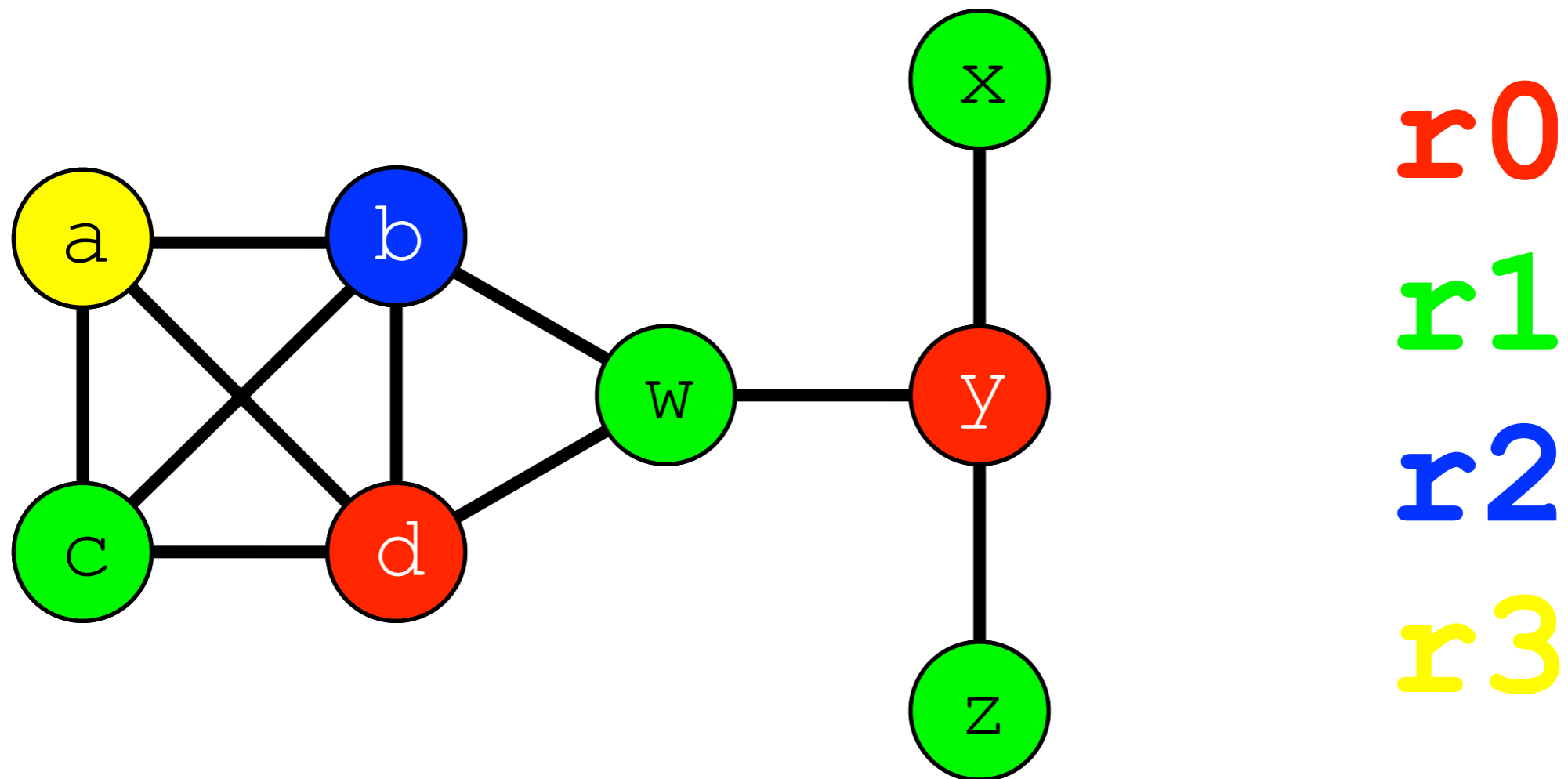
Algorithm

- Choose a vertex (i.e. virtual register) which has the least number of incident edges (i.e. clashes).
- Remove the vertex and its edges from the graph, and push the vertex onto a LIFO stack.
- Repeat until the graph is empty.
- Pop each vertex from the stack and colour it in the most conservative way which avoids the colours of its (already-coloured) neighbours.

Algorithm

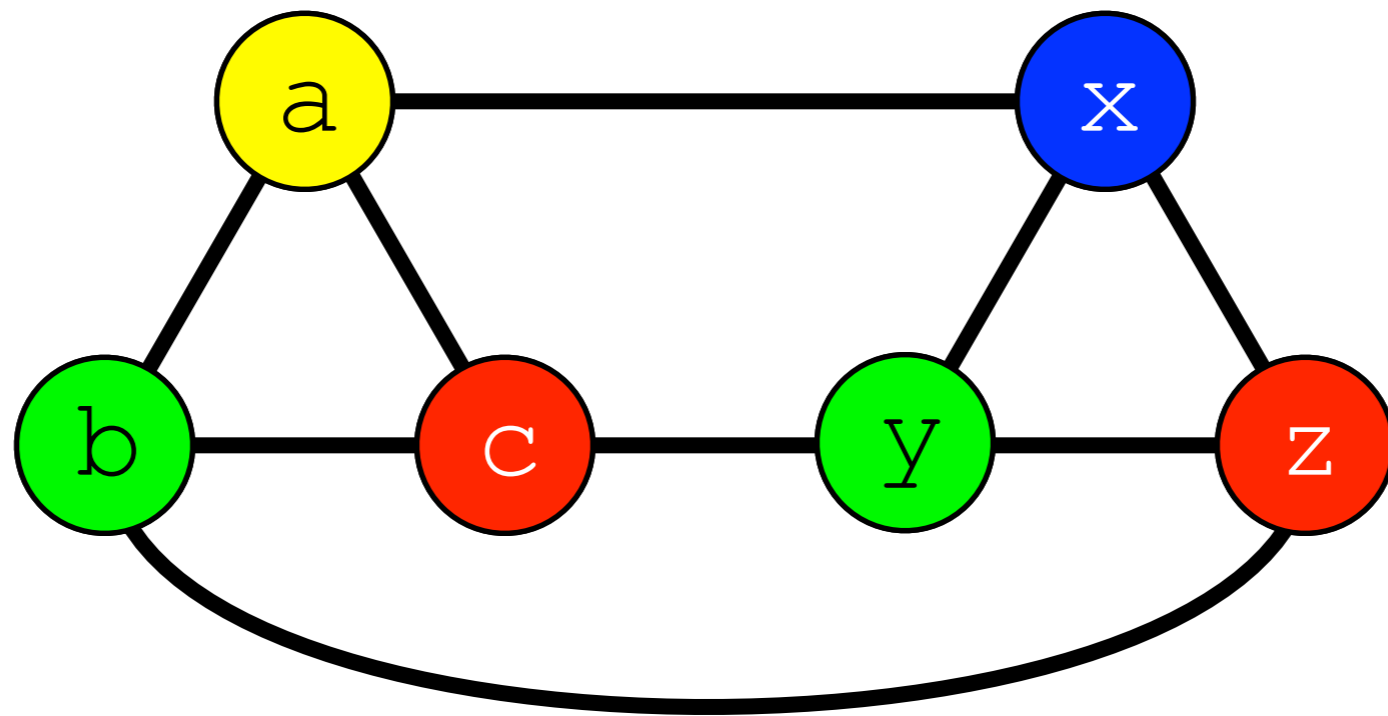


Algorithm



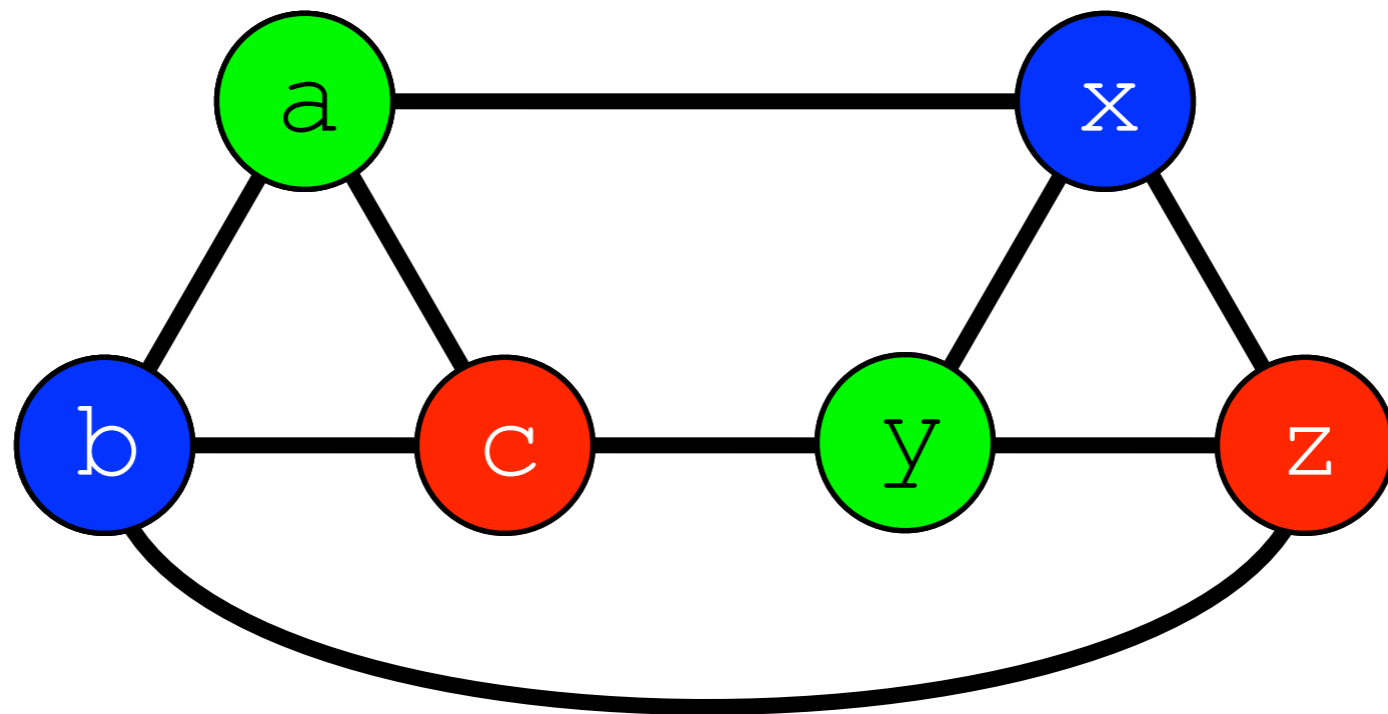
Algorithm

Bear in mind that this is only a heuristic.



Algorithm

Bear in mind that this is only a heuristic.



A better (more minimal) colouring may exist.

Spilling

This algorithm tries to find an approximately minimal colouring of the clash graph, but it assumes new colours are always available when required.

In reality we will usually have a finite number of colours (i.e. architectural registers) available; how should the algorithm cope when it runs out of colours?

Spilling

The quantity of architectural registers is strictly limited, but it is usually reasonable to assume that fresh memory locations will always be available.

So, when the number of simultaneously live values exceeds the number of architectural registers, we may *spill* the excess values into memory.

Operating on values in memory is of course much slower, but it gets the job done.

Spilling

```
ADD a, b, c
```

vs.

```
LDR t1, #0xFFA4
```

```
LDR t2, #0xFFA8
```

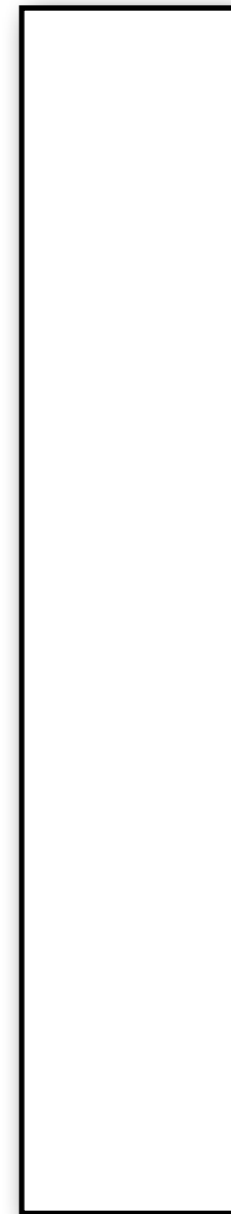
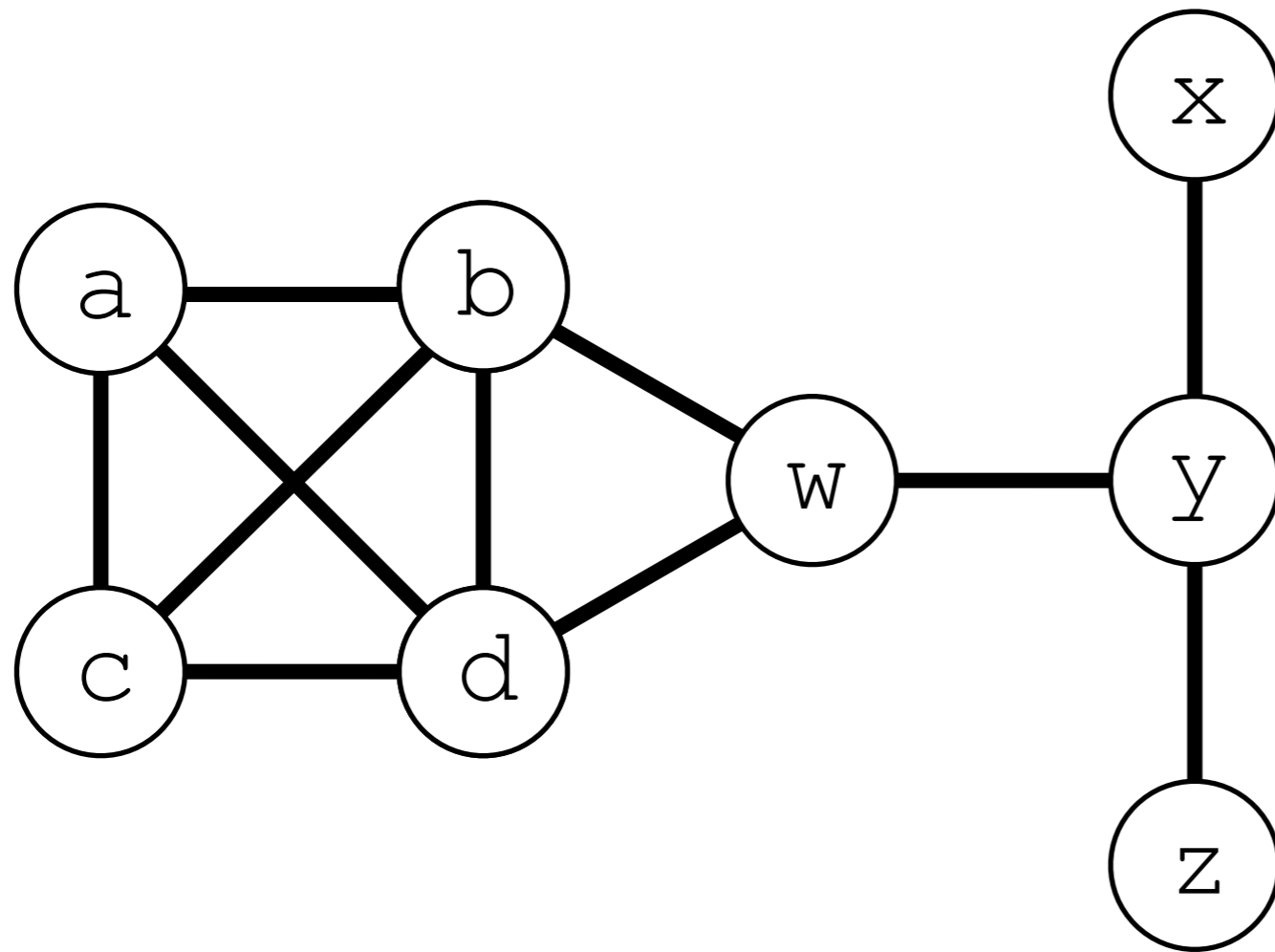
```
ADD t3, t1, t2
```

```
STR t3, #0xFFA0
```

Algorithm

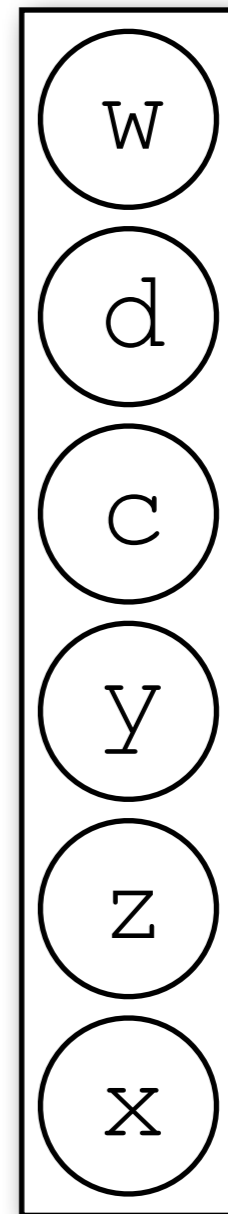
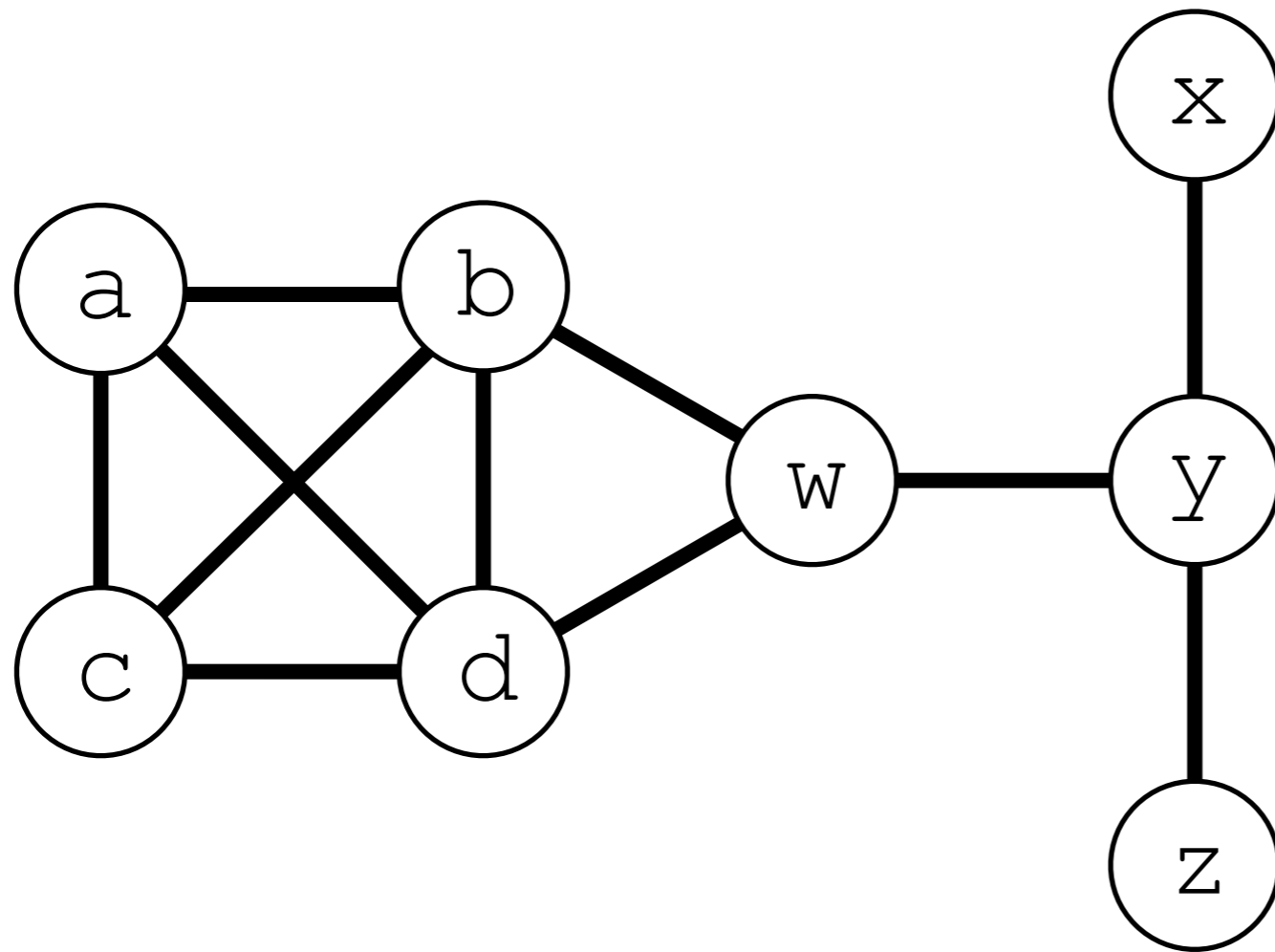
- Choose a vertex with the least number of edges.
- If it has fewer edges than there are colours,
 - remove the vertex and push it onto a stack,
 - otherwise choose a register to spill — e.g. the least-accessed one — and remove its vertex.
- Repeat until the graph is empty.
- Pop each vertex from the stack and colour it.
- Any uncoloured vertices must be spilled.

Algorithm

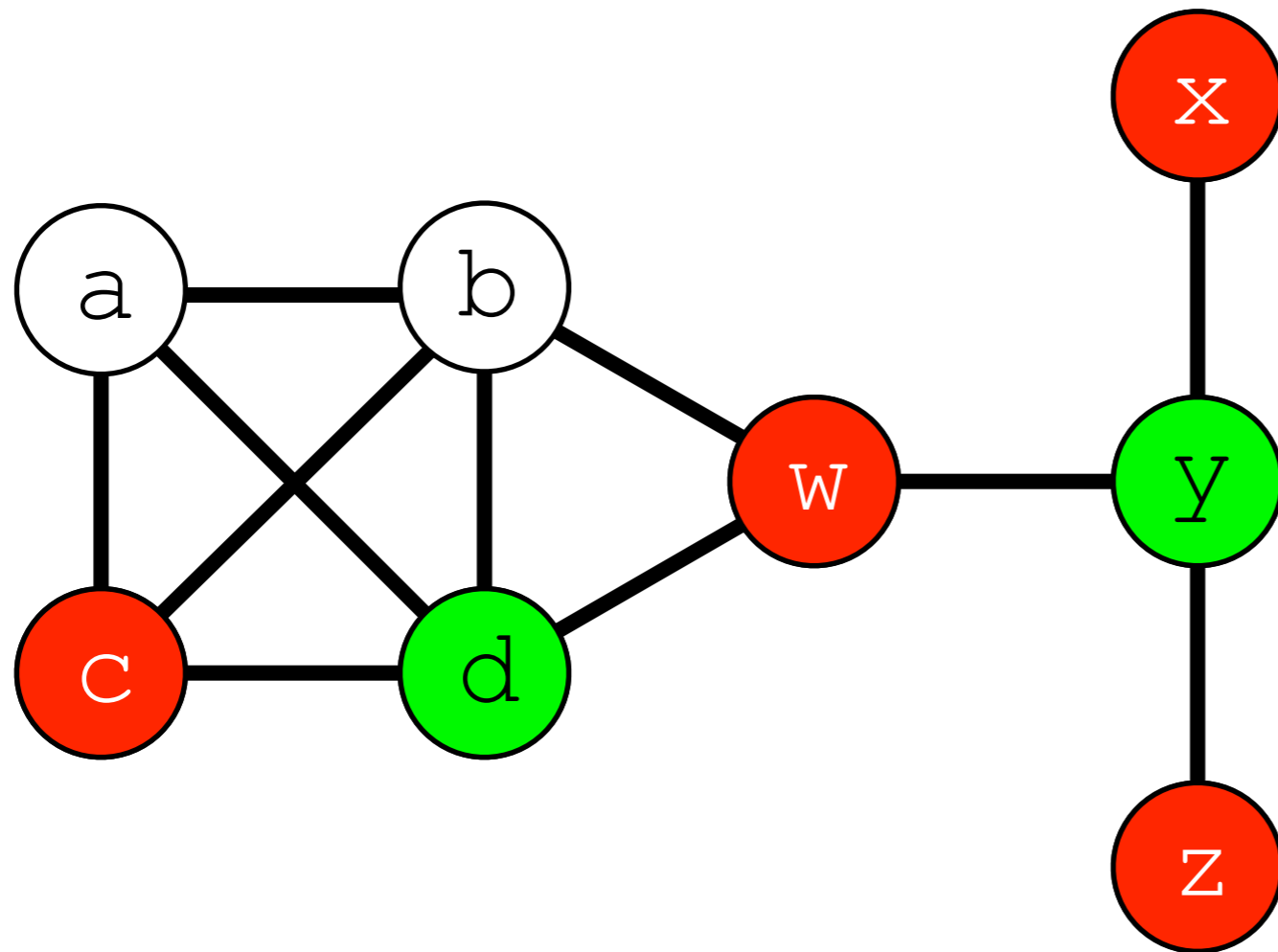


a: 3, b: 5, c: 7, d: 11, w: 13, x: 17, y: 19, z: 23

Algorithm



Algorithm



r0
r1

a and b
spilled to memory

a: 3, b: 5, c: 7, d: 11, w: 13, x: 17, y: 19, z: 23

Algorithm

Choosing the right virtual register to spill will result in a faster, smaller program.

The static count of “how many accesses?” is a good start, but doesn’t take account of more complex issues like loops and simultaneous liveness with other spilled values.

One easy heuristic is to treat one static access inside a loop as (say) 4 accesses; this generalises to 4^n accesses inside a loop nested to level n .

Algorithm

“Slight lie”: when spilling to memory, we (normally) need one free register to use as temporary storage for values loaded from and stored back into memory.

If any instructions operate on two spilled values simultaneously, we may need *two* such temporary registers to store both values.

So, in practise, when a spill is detected we may need to restart register allocation with one (or two) fewer architectural registers available so that these can be kept free for temporary storage of spilled values.

Algorithm

When we are popping vertices from the stack and assigning colours to them, we sometimes have more than one colour to choose from.

If the program contains an instruction “MOV a, b” then storing a and b in the same arch. register (as long as they don't clash) will allow us to delete that instruction.

We can construct a *preference graph* to show which pairs of registers appear together in MOV instructions, and use it to guide colouring decisions.

Non-orthogonal instructions

We have assumed that we are free to choose architectural registers however we want to, but this is simply not the case on some architectures.

- The x86 `MUL` instruction expects one of its arguments in the `AL` register and stores its result into `AX`.
- The VAX `MOVCL3` instruction zeroes `r0`, `r2`, `r4` and `r5`, storing its results into `r1` and `r3`.

We must be able to cope with such irregularities.

Non-orthogonal instructions

We can handle the situation tidily by pre-allocating a virtual register to each of the target machine's arch. registers, e.g. keep v_0 in r_0 , v_1 in r_1 , ..., v_{31} in r_{31} .

When generating intermediate code in normal form, we avoid this set of registers, and use new ones (e.g. v_{32} , v_{33} , ...) for temporaries and user variables.

In this way, each architectural register is explicitly represented by a unique virtual register.

Non-orthogonal instructions

We must now do extra work when generating intermediate code:

- When an instruction requires an operand in a specific arch. register (e.g. x86 MUL), we generate a preceding MOV to put the right value into the corresponding virtual register.
- When an instruction produces a result in a specific arch. register (e.g. x86 MUL), we generate a trailing MOV to transfer the result into a new virtual register.

Non-orthogonal instructions

If (hypothetically) ADD on the target architecture can only perform $r0 = r1 + r2$:

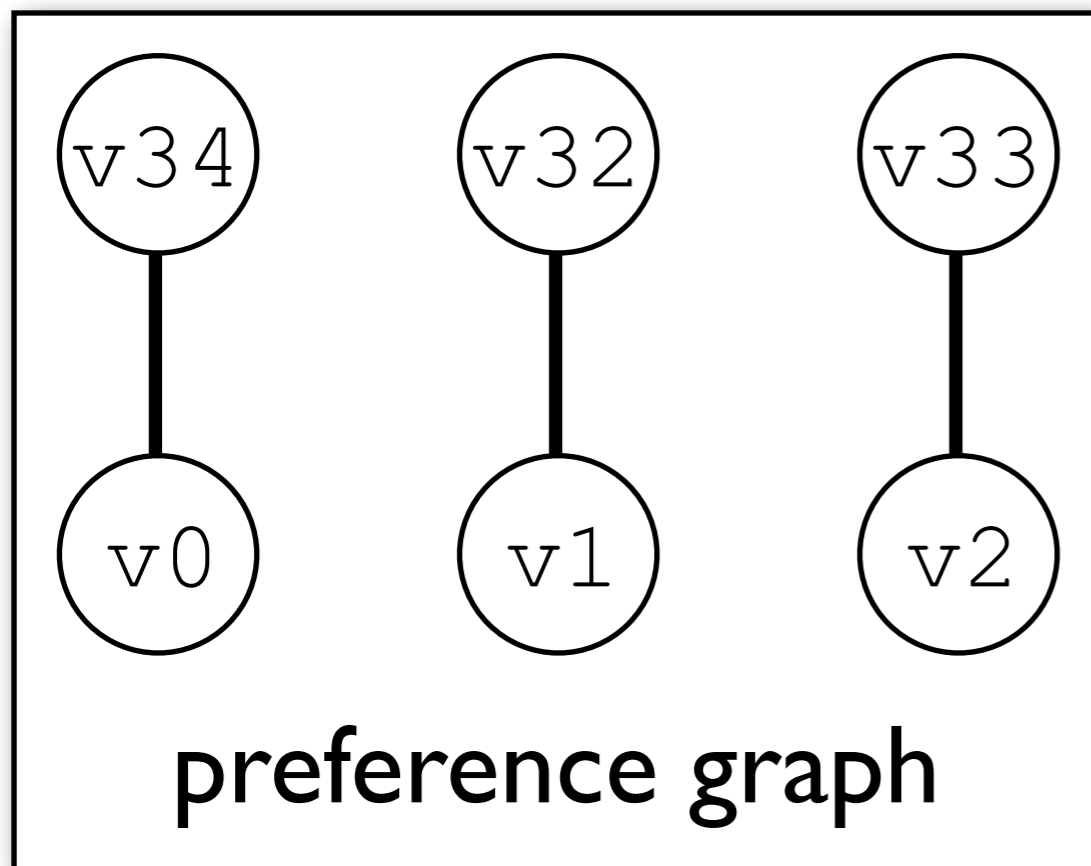
```
x = 19;  
y = 23;  
z = x + y;
```



```
MOV v32, #19  
MOV v33, #23  
MOV v1, v32  
MOV v2, v33  
ADD v0, v1, v2  
MOV v34, v0
```

Non-orthogonal instructions

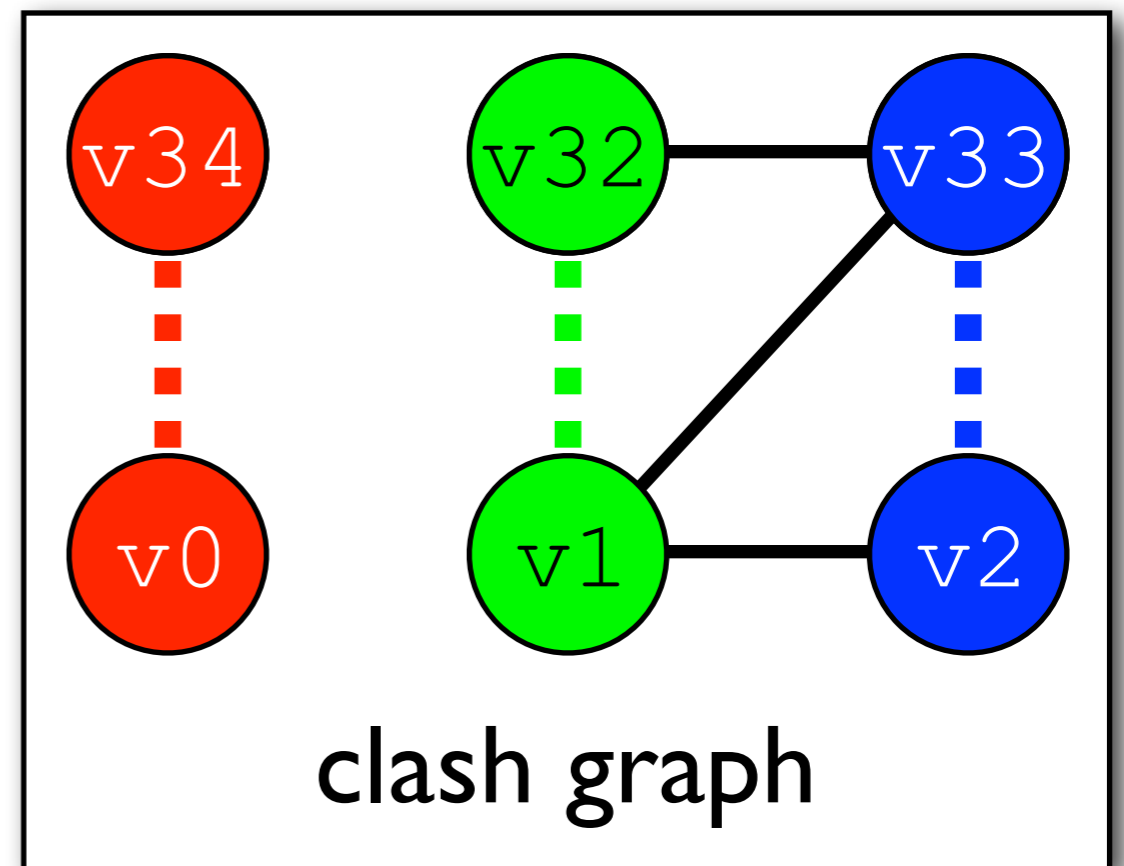
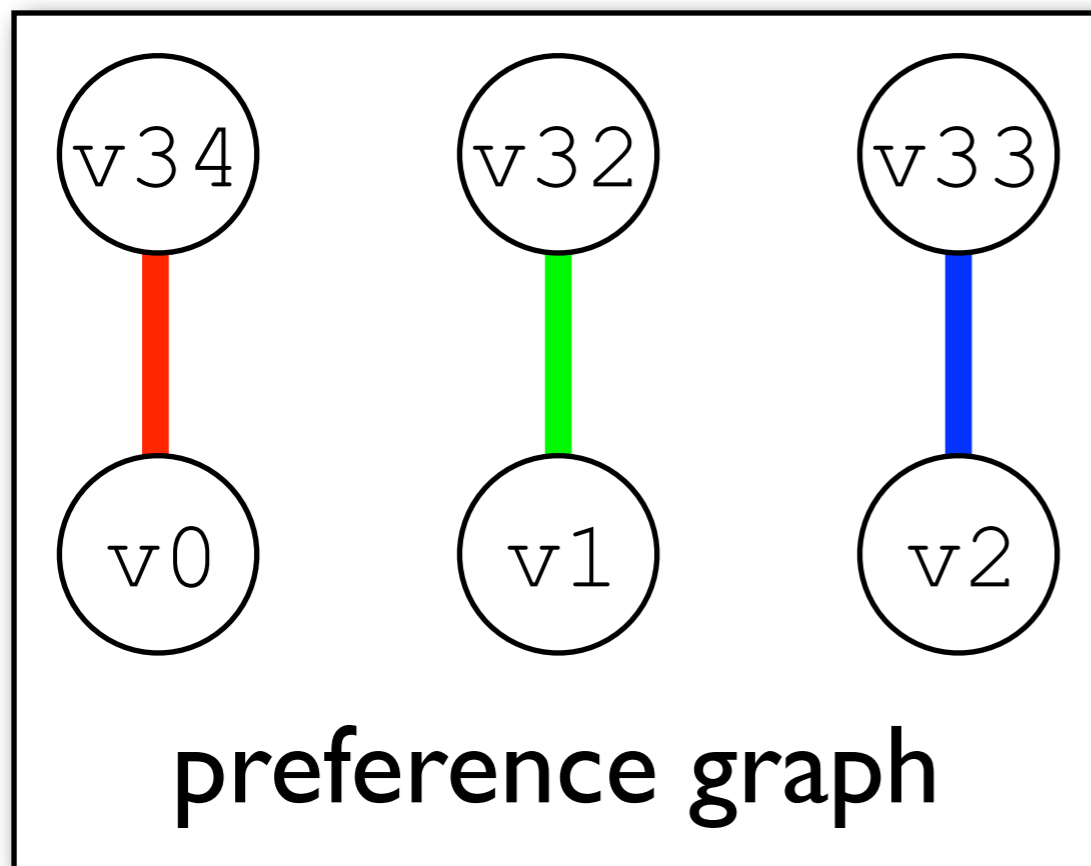
This may seem particularly wasteful, but many of the `MOV` instructions will be eliminated during register allocation if a preference graph is used.



```
MOV v32, #19
MOV v33, #23
MOV v1, v32
MOV v2, v33
ADD v0, v1, v2
MOV v34, v0
```

Non-orthogonal instructions

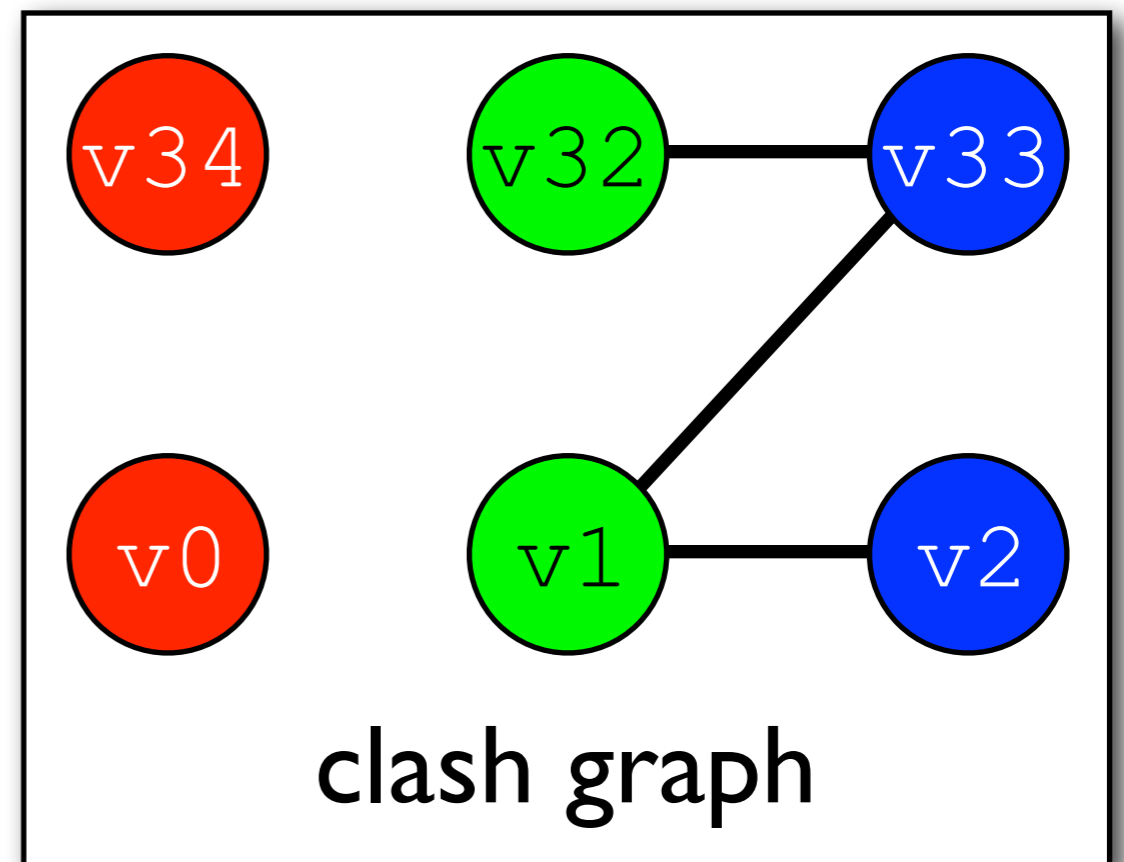
This may seem particularly wasteful, but many of the `MOV` instructions will be eliminated during register allocation if a preference graph is used.



Non-orthogonal instructions

This may seem particularly wasteful, but many of the MOV instructions will be eliminated during register allocation if a preference graph is used.

```
MOV  r1, #19
MOV  r2, #23
MOV  r1, r1
MOV  r2, r2
ADD  r0, r1, r2
MOV  r0, r0
```



Non-orthogonal instructions

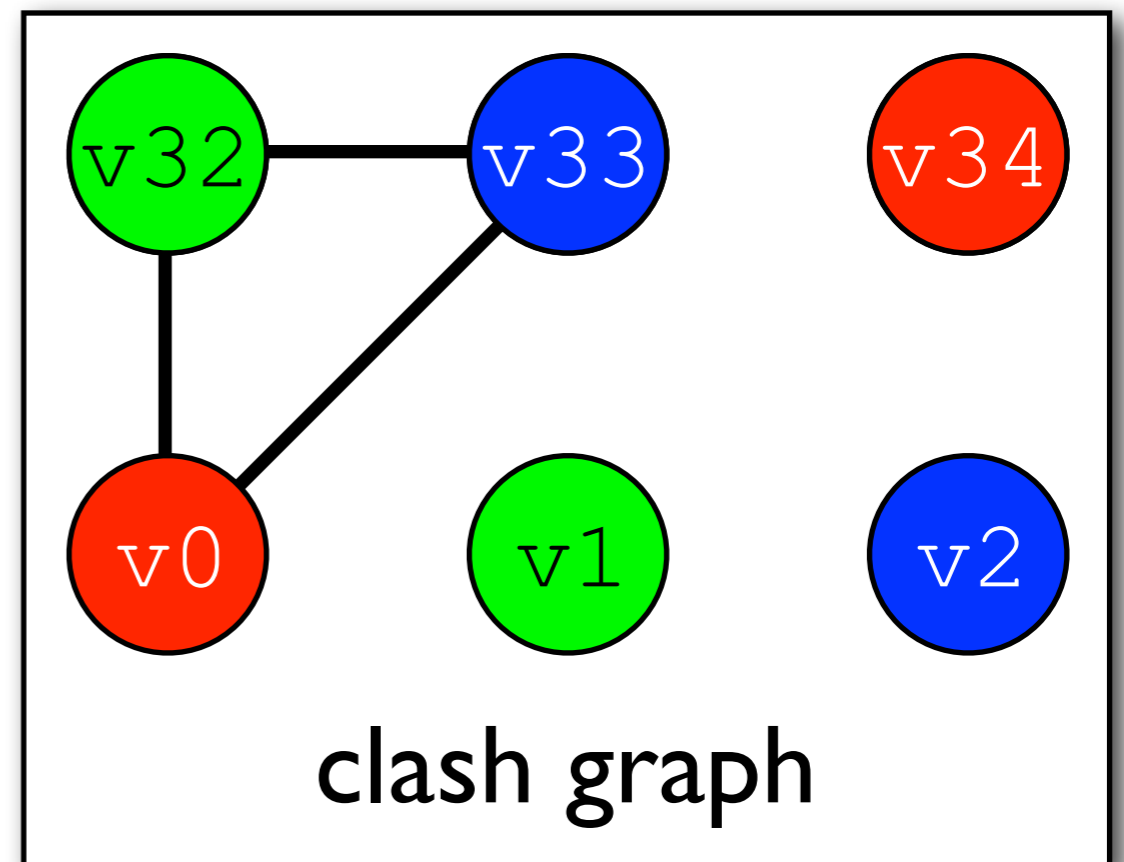
And finally,

- When we know an instruction is going to corrupt the contents of an architectural register, we insert an edge on the clash graph between the corresponding virtual register and all other virtual registers live at that instruction — this prevents the register allocator from trying to store any live values in the corrupted register.

Non-orthogonal instructions

If (hypothetically) MUL on the target architecture corrupts the contents of `r0`:

```
MOV  r1, #6  
MOV  r2, #7  
MUL  r0, r1, r2  
⋮
```



Procedure calling standards

This final technique of synthesising edges on the clash graph in order to avoid corrupted registers is helpful for dealing with the procedure calling standard of the target architecture.

Such a standard will usually dictate that procedure calls (e.g. `CALL` and `CALLI` instructions in our 3-address code) should use certain registers for arguments and results, should preserve certain registers over a call, and may corrupt any other registers if necessary.

Procedure calling standards

On the ARM, for example:

- Arguments should be placed in `r0-r3` before a procedure is called.
- Results should be returned in `r0` and `r1`.
- `r4-r8`, `r10` and `r11` should be preserved over procedure calls, and `r9` might be depending on the platform.
- `r12-r15` are special registers, including the stack pointer and program counter.

Procedure calling standards

Since a procedure call instruction may corrupt some of the registers ($r0-r3$ and possibly $r9$ on the ARM), we can synthesise edges on the clash graph between the corrupted registers and all other virtual registers live at the call instruction.

As before, we may also synthesise MOV instructions to ensure that arguments and results end up in the correct registers, and use the preference graph to guide colouring such that most of these MOVs can be deleted again.

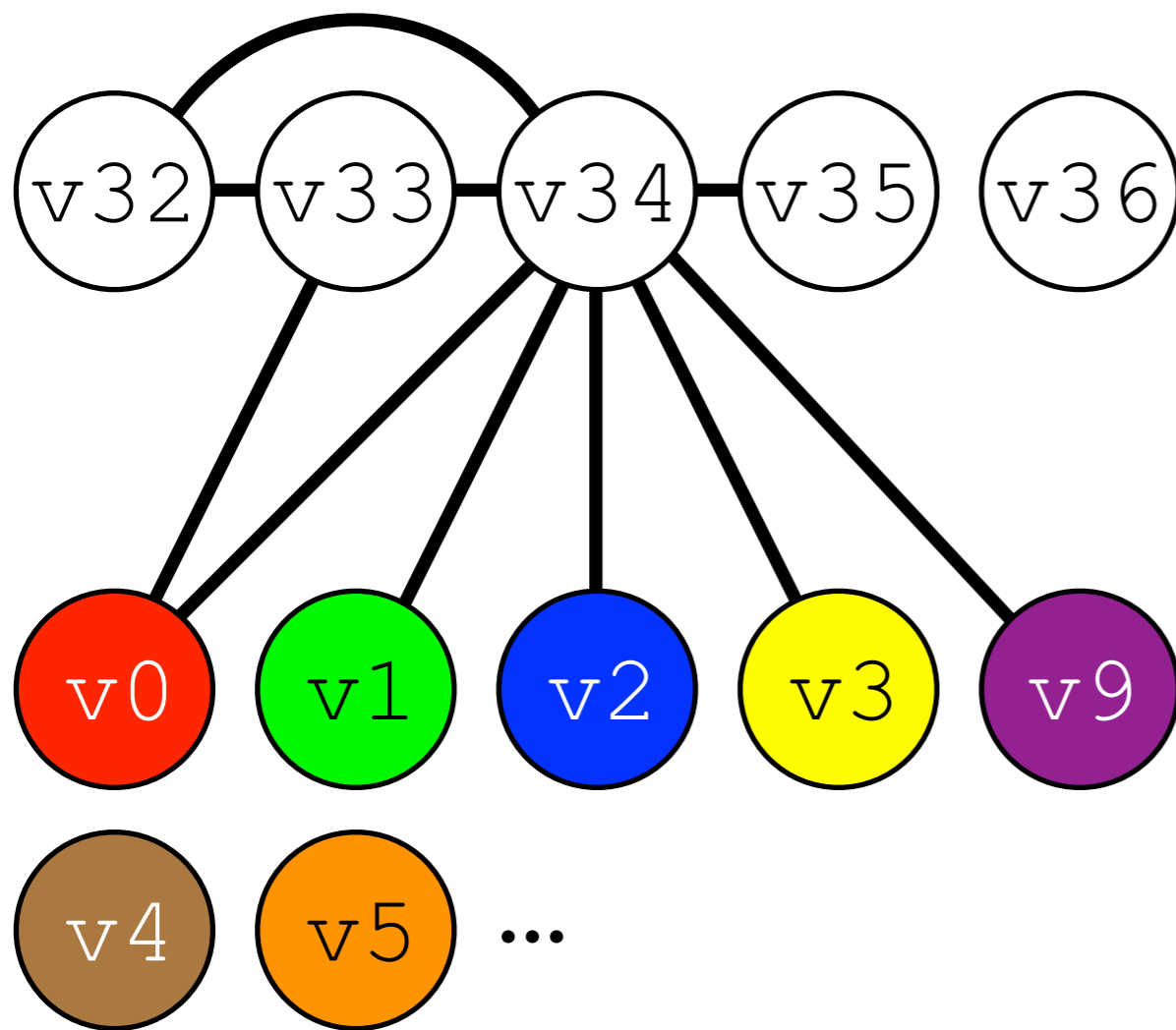
Procedure calling standards

```
x = 7;  
y = 11;  
z = 13;  
a = f(x, y) + z;
```



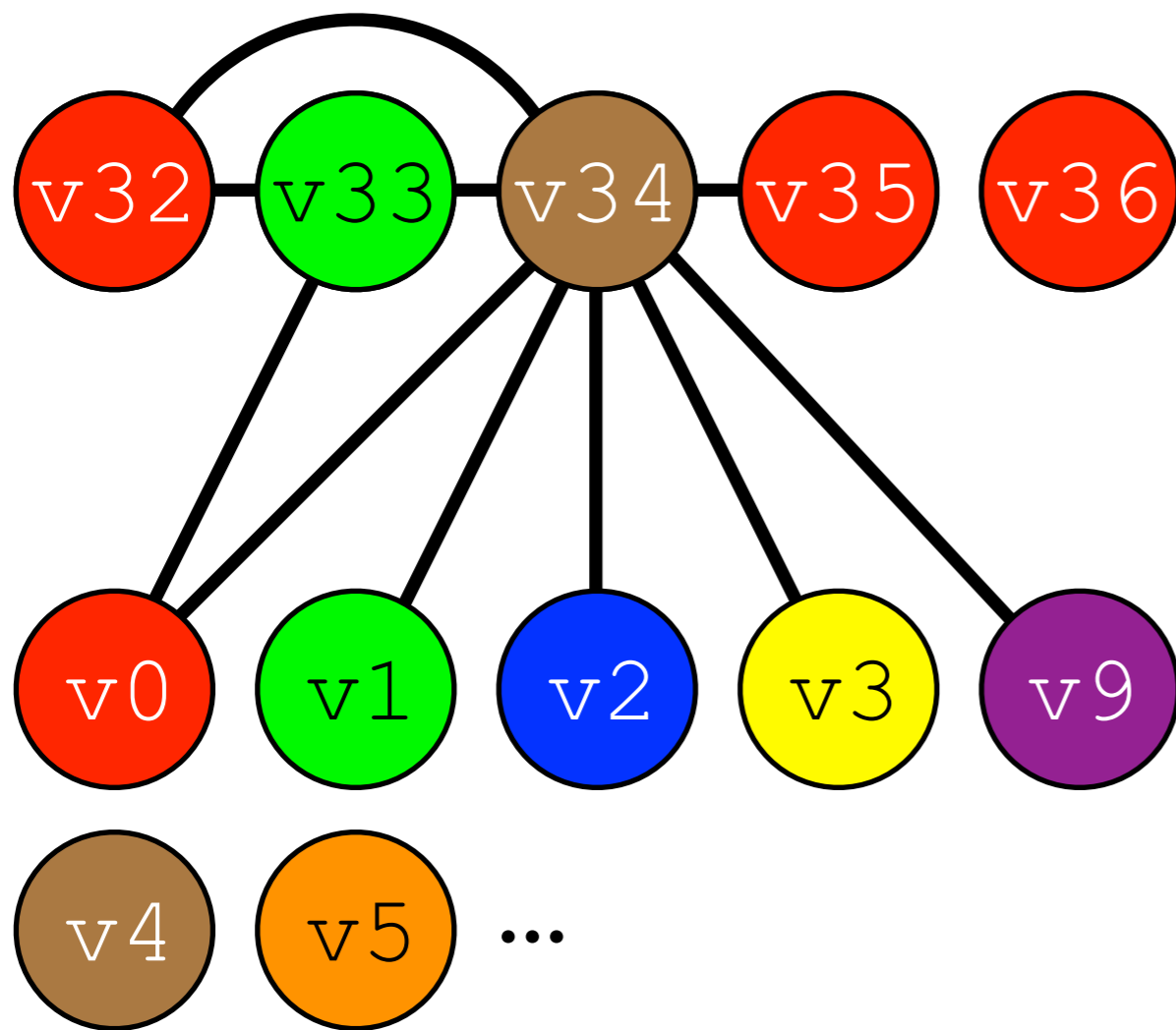
```
MOV v32, #7  
MOV v33, #11  
MOV v34, #13  
MOV v0, v32  
MOV v1, v33  
CALL f  
MOV v35, v0  
ADD v36, v34, v35
```

Procedure calling standards



```
MOV    v32, #7
MOV    v33, #11
MOV    v34, #13
MOV    v0, v32
MOV    v1, v33
CALL   f
MOV    v35, v0
ADD    v36, v34, v35
```


Procedure calling standards



```
MOV  r0, #7
MOV  r1, #11
MOV  r4, #13
MOV  r0, r0
MOV  r1, r1
CALL f
MOV  r0, r0
ADD  r0, r4, r0
```

Summary

- A register allocation phase is required to assign each virtual register to an architectural one during compilation
- Registers may be allocated by colouring the vertices of a clash graph
- When the number of arch. registers is limited, some virtual registers may be spilled to memory
- Non-orthogonal instructions may be handled with additional MOVs and new edges on the clash graph
- Procedure calling standards also handled this way